ORIGINAL CONTAINS
COLOR ILLUSTRATIONS

AMR on the CM-2

Marsha J. Berger and Jeff Saltzman

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 311, Columbia, MD 244, (301)730-2656

Work reported herein was supported in part by the NAS Systems Division of NASA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035.

AMR on the CM-2

Marsha J. Berger
Courant Institute
251 Mercer Street
New York, NY 10012

Jeff S. Saltzman
Los Alamos National Laboratory
MS B265
Los Alamos, NM 87545

Abstract

We describe the development of a structured adaptive mesh algorithm (AMR) for the CM-2. We develop a data layout scheme that preserves locality even for communication between fine and coarse grids.On 8K of a 32K machine we achieve performance slightly less than 1 CPU of the Cray Y-MP. We apply our algorithm to an inviscid compressible flow problem.

1 Introduction

Local Adaptive Mesh Refinement (AMR) is an algorithm that can efficiently compute complex flow fields where only a small fraction of the computational domain needs to be resolved. Its success has been demonstrated over the last ten years in [1, 4, 3, 15, 20, 21]. A natural question is whether this approach is still viable on massively parallel architectures. In particular, can we still take advantage of local regularity of the grids? Can the dynamic and adaptive character of AMR be maintained while balancing the load on a fine grained data parallel/SIMD machine such as the CM-2? A new issue not found on serial architectures is minimizing inter-grid communication forced by the distributed memory.

In the last few years several adaptive unstructured mesh codes have been parallelized [7, 22, 17]. Unstructured mesh methods tend to have much different overheads and efficiencies than structured mesh codes. Their data structures are lists of elements/edges, leading to more words of storage per node than structured methods use. However, this generality and flexibility lends itself to a natural extension to both coarse grained and fine grained parallel architectures. The issues in the parallelization of unstructured methods are the creation of subdomain partitions, and the mapping of the partitions onto individual processor nodes to minimize global communication and balance the load. The communication costs seem to be the major source of inefficiency in these codes, even for non-adaptive unstructured parallel codes. For example, Barth and Hammond [14] report 50% of the run time on the CM-2 is spent in communications tasks. The free-Lagrange code X3D [7] has been measured to spend 93% of its time in communication [18].

In contrast, very little work on structured meshes has been done. Gropp and Keyes have developed interesting algorithms for adaptive elliptic equations on semi-structured meshes [13]. Similar looking meshes are found in the parallel AFAC algorithms of [19]. Dynamically adaptive parallel algorithms for hyperbolic equations on quad tree data structures have been developed in [5, 6]. This work uses a coarse grained model of parallel computation, with the parallelization coming from different ways of traversing the tree.

Most other work on parallelization of adaptive structured meshes, to our knowledge, has targeted AMR, and uses coarse-grained parallelization on small numbers of processors. Since the data structures in AMR keep track of entire grids, rather than individual grid points, a natural approach here is to distribute the grids to different processors. Berger [2] has done this on a shared memory Cray X-MP4/16, and Crutchfield [10, 11] on a 32 node BBN TC2000. Neither of these approaches can take advantage of massively parallel computers. Our largest 3D application so far has used on the order of 500 grids at a given time. Even allowing for future applications with several thousand grids, this coarse grained approach would not scale well for a machine with several thousand processors or more.

We have developed a data parallel implementation of a 2-D AMR code for the CM-2. (For a discussion of the CM-2 architecture see [16].) In this approach the individual points in a grid are distributed to processors. The key idea in our strategy is the data layout. We map the points of grids on different levels to minimize intergrid global communication and preserve locality. In addition, the serial algorithm was modified in several ways, in particular, we have restricted the adaptive grid patches to be a fixed size. We compare the efficiency of our implementation on an 8K CM-2 with a functionally equivalent implementation on a Cray Y-MP. We measure that the ratio of integration time to total CPU time of a typical run approaches 75% on the CM-2 while this ratio on the Cray Y-MP is closer to 85%. We also measure the grind time of our integration procedure, defined as the time to update one grid point one timestep. This is roughly equivalent to the Cray Y-MP time. Thus, our performance on an 8K CM-2 is slightly less than one head of a Cray Y-MP.

Our work was hampered by inadequacies in the CM slicewise Fortran compiler. There are additional opportunities for a coarser grained parallelism to be used on top of the fine grained parallelism, within the data parallel framework. Unfortunately, we could not exploit it because version 1.1 of the slicewise compiler does not parallelize/vectorize the outer loop of a serial dimension. However, even without this additional parallelism we have demonstrated the viability of the fine grained approach using the current compiler technology. Future releases of the CM Fortran compiler may lead to further exploitation of parallelism, as will extensions of our work to the CM-5. We hope that future releases also include richer array section constructions, and more flexible alignment and layout directives; the lack of which led to considerable programming headache during the course of this work.

The paper is organized as follows. Section 2 gives a brief overview of the serial AMR algorithm. Section 3 describes the data parallel version of the algorithm. The important points here are the data layout and grid generation. Section 4 discusses implementation details pertaining to the CM-2. Section 5 gives timings of the performance of the integrator alone and the overall AMR algorithm. We describe the numerical simulation of laser trenching in an integrated circuit substrate using a 2-D gas dynamics approximation of the physics in a simple geometry. In the conclusion we have several recommendations and a discussion of the parallelism we currently cannot exploit and its potential impact on the performance of the algorithm.

2 Overview of the AMR Algorithm

The AMR algorithm uses a nested sequence of rectangular grids to approximate the solution to a partial differential equation. The state variables are cell-centered, and an explicit finite volume scheme updates these values by computing fluxes at cell edges:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - \Delta t\Big[\frac{(F_{i+1/2,j} - F_{i-1/2,j})}{\Delta x} - \frac{(G_{i,j+1/2} - G_{i,j-1/2})}{\Delta y}\Big]$$

When the solution resolution is insufficient, rather than refining a single grid cell at a time, rectangular fine grid patches are generated to cover those cells that need additional refinement. These grid patches have their own solution storage, with minimal storage overhead needed to describe the location and size of the grid itself. Grids are properly nested and aligned with each other, i.e. for every cell in a level l grid there is at least one level $l-1$ cell surrounding it in all directions, although these coarse cells may belong to different grids. The grids are not rotated with respect to each other. Note that a fine grid can have more than one parent grid (see figure 1).

Each grid level has its own time step. Typically, if a grid is refined by a factor of 4, the time step is refined by a factor of 4 because of time step stability restrictions. The integration procedure on such a grid hierarchy then proceeds recursively: integrate on the coarse grid (ignoring fine grids); then use the coarse grid values with space time interpolation to provide boundary conditions for fine grids so that they, too, may be advanced in time. The algorithm is recursive in that the fine grid is in turn used in advancing still finer grids.

There are four separate components to the AMR algorithm that together generate this adaptive mesh hierarchy and advance the solution. For a complete description of the AMR algorithm see [1, 3]. The *error estimator* decides where the solution accuracy on a given grid level is insufficient and tags those grid cells as needing refinement. The *grid generator* creates mesh patches that cover all the flagged points. It takes as input the set of tagged points from the error estimator, and outputs a set of grid patches that together cover all the cells needing refinement. The inter-grid communication happens in the following two components. The *interpolation* routines initialize a solution on a new fine mesh, from

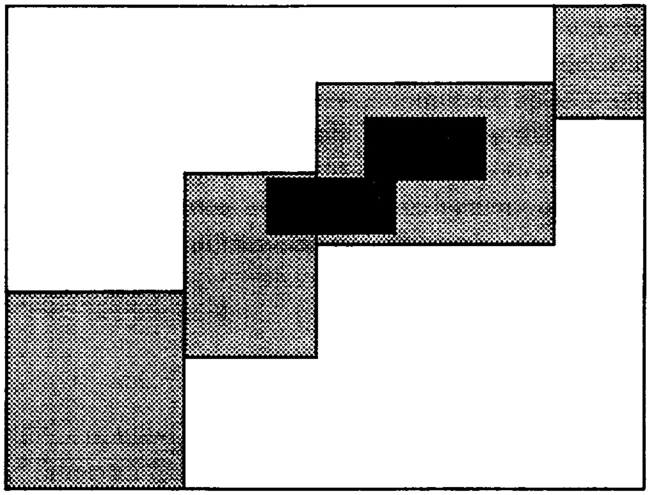


Figure 1: There is a single level 1 grid, level 2 has 4 grids and level 3 has 2 grids in this hierarchy.

either old fine meshes (injection) or coarser grids (interpolation), and they provide enough boundary values for fine meshes so that the integration stencil can be used at every interior point in a fine grid. The ***flux correction*** routine insures conservation at grid interfaces by modifying those coarse grid cells adjacent to a fine grid. We strictly enforce the condition that the flux out of a coarse grid cell during a single coarse time step equals the flux into the adjacent fine grid cells over all the corresponding fine time steps. This component also includes what we call ***updating;*** the fine grid cells update the coarse grid cell "underneath" using a conservative, volume weighted average of the fine grid values.

We typically use a high-order Godunov method to integrate a system of conservation laws [8, 9]. The integrator can be operator split or unsplit, the system of equations can be augmented by passively advected quantities, such as for multiple species, and the AMR shell doesn't change.

3 The SIMD AMR Algorithm

In our data parallel version of the AMR algorithm, individual grid points rather than entire grids are mapped to processors. We make the restriction that all grid patches are a fixed size. This allows us to design a data layout scheme for mapping points to processors to minimize inter-grid global communication and preserve locality. This is the key idea in our approach.

The grid size restriction is motivated by the following considerations. When we first considered the mapping of grids to processors, we thought several grids of varying sizes

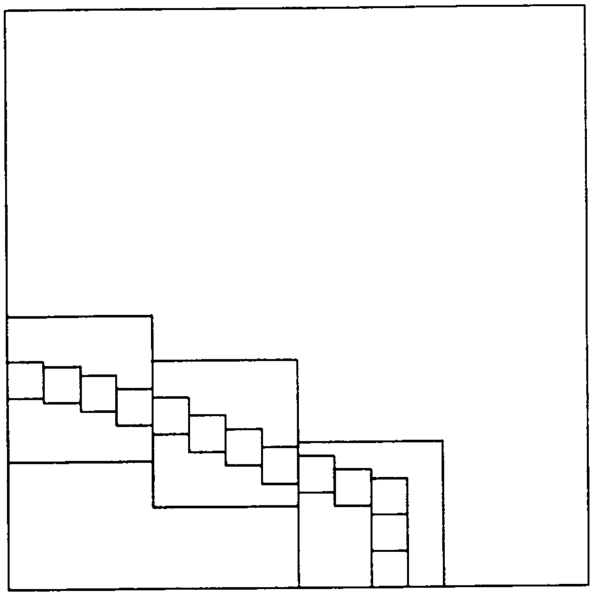


Figure 2: Three levels of grids are shown; each has the same number of grid points.

might be integrated at one time. A multi-dimensional type of bin packing could be used to group the fine grid patches to fill up a two-dimensional view of the machine. However, the CM-2 at NASA Ames Research Center has 32K processors. One quarter of the machine (the typical amount attached to a sequencer) has 8K processors, but only 256 floating point units. This corresponds to a 16 by 16 mesh. To keep the floating point pipes full requires a minimum of 4 grid points per processor. Hence a single 32 by 32 mesh can use one quarter of the machine and a single 64 by 64 grid patch can use the whole machine. With typical grid sizes in this range, a natural choice was to keep all grids the same size. In practice, on the order of 16-32 points per processor are usually needed for peak machine performance. Therefore, what fixed size is chosen is extremely important to the performance of the algorithm. Can we find a tile size small enough to use effectively in an adaptive setting, yet large enough to be integrated efficiently? In sections 4 and 5 we discuss the ramifications and efficiencies of this decision. Figure 2 shows an example of three levels of grids, each has the same number of grid points but smaller and smaller mesh widths, so they occupy a decreasing amount of physical space.

Given this restriction of fixed sizes grids (either 32 or 64 in each dimension), we have designed a data layout scheme for both the coarse and fine grids that preserves locality and minimizes communication. The layout is best described using the following indexing notation. We describe it using one space dimension. Higher dimensions use tensor products of the one-dimensional case.

Memory location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Fine Cell Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Coarse Cell Index	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16
Fine Cell Index	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Fine Cell Index	33	34	35	36	37	38	39	40	25	26	27	28	29	30	31	32

Table 1: Illustration of the mapping from grid points to memory location for the grids in figure 3.

Each coarse grid cell in the computational domain is given a unique index. If the computational domain consists of a single coarse grid with tile size 32, the cells are numbered from 1 to 32. If there are 2 coarse grids in the computational domain, adjacent to each other, the first is numbered from 1 to 32, and the second from 33 to 64. Suppose now there is a single coarse grid, and it is completely covered by fine grid patches. If the refinement ratio is 4, then 4 fine grids make up the computational domain. The i^{th} fine grid will be numbered from $(i-1)*32+1$ to $i*32$, corresponding to coarse grid cells $(i-1)*32/4+1$ to $(i-1)*32/4+8$. Of course, the fine grids do not have to start at location that are multiples of 32; for example a fine grid can be numbered from 25 to $25+32-1=56$.

Given this numbering convention, we can describe the two different layout strategies for the fine and coarse grids. (More precisely, the grid patch is mapped to a two-dimensional view of memory using the usual CM compiler default mapping, but we "interpret" it in a different way). Suppose for simplicity the tile size is 16. The finest grids are mapped to memory so that cell i is in memory location $i \bmod 16$, or more precisely, $(i-1) \bmod 16+1$. In other words, the grids are periodically wrapped as they are mapped to the CM memory so that adjacent cells from adjacent fine grids are in adjacent memory location. This keeps the injection operation from grids at the same level completely local.

The coarse grid layout is the complicated one. Suppose the refinement ratio is 4. Then cell 1 on the coarse grid corresponds to cells 1 through 4 on the fine level, coarse cell 2 to fine cells 5 through 8, etc. To keep the coarse/fine grid communication local, the rows and columns of the coarse grid are permuted so that no matter where the fine or coarse grid is located, corresponding cells are within 4 of each other. This enables fast data movement using nearest neighbor NEWS wires. For a single coarse grid as shown in figure 3, the grid point to memory mapping is shown in table 1. Note that even for the second fine grid, the corresponding coarse points are "nearby". For example, point 17 on the fine grid is corresponds to cell 5 on the coarse grid, and is one memory location away. This same correspondence holds regardless of where the fine grid is located, or how many coarse grids there are.

This coarse grid permutation is derived by sequentially distributing the coarse grid points, skipping 4 memory locations between points (for a refinement ratio of 4). When the

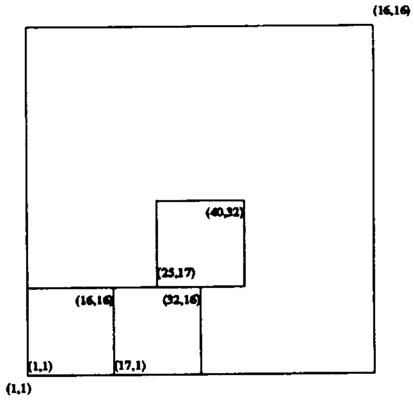


Figure 3: One coarse grid with three refined grids, corresponding to the memory layout in table 1. for a tile size of sixteen

end is reached, wrap around to the next available memory location from the beginning.

As often happens, an improvement to the serial AMR algorithm for proper nesting was discovered by rethinking the algorithm for the SIMD implementation. The proper nesting requirement for fine grids can be simply checked using a "domain calculus" with simple logical operations and a bitmap of the domain. The description of the set of all grids at a given level is simple to compute regardless of how complicated the domain at that level is. Given a domain at a certain level, properly nested subgrids must be one cell away from the boundary of the domain. Again this is simply computed with some shifts and logical operations. Previous (serial) implementations used complicated testing along fine grid boundaries to verify proper nesting.

The grid generation component of the AMR algorithm was greatly simplified by the use of fixed sized tiles. The serial version uses a pattern recognition algorithm to find "edges" in the tagged points; these edges form the edges of the new grid patches. (See [1] for details). Although rather sophisticated, this grid generation algorithm used negligible run time, and usually produced grids with approximately 80% efficiency, i.e., 80% of the points in the new grids were tagged, only 20% were additionally included to keep grids rectangular. The new grid generator simply tiles the smallest rectangular region bounding all the tagged cells using the fixed size tiles. It may happen that tiles have no tagged points underneath them, (for example, if the tagged points form a large circle and the tile is inside the circle). In this case the tile is deleted. This unsophisticated approach produces refined grids around the 50% efficiency level. Also, the use of fixed size tiles and the requirement of proper nesting may necessitate the use of overlapping grids. This rarely happens in practise, although it does slightly increase programming complexity.

In the original work of [3], Richardson extrapolation was used to estimate the error in the computation. This however had to be augmented by additional procedures depending

	Two Dimensions		Three Dimensions	
PatchSize	2 Ghost Cells	4 Ghost Cells	2 Ghost Cells	4 Ghost Cells
32	77%	56%	67%	42%
64	88%	77%	82%	67%

Table 2: Percentage of interior cells as a function of tile size

on the type of fluid flow being computed, for example, density gradients or compressibility and divergence were computed. We use only the latter, more ad hoc estimates in this work.

4 Implementation Details

The implementation decision with the most impact on performance concerned the treatment of ghost cells. To avoid special boundary stencils, each grid is surrounded by the additional number of points needed to apply the same finite volume stencil everywhere. Our simplified version of the high-order Godunov method uses 3 points to the side. These so-called ghost cells, or dummy cells, are obtained from adjacent fine grids or interpolated from coarser grids. Every grid contains space for these ghost cells along with the regular interior cells. For a grid of size 32, if there are 3 ghost cells on each side, only 26 cells are left for what we call the "real" interior grid cells. Since 26 is not divisible by 4 (our usual refinement ratio), in fact we must allocate 4 ghost cells on each side (all 4 are actually used in the more sophisticated version of the integrator).

Table 2 shows the fraction of interior cells as a function of tile size for 2 and 3 space dimensions. As can be seen, in 3 dimensional calculations for 32 sized tiles the fraction drops below 50%. In 2 space dimensions, 32 sized tiles is a possibility, although there are other reasons to prefer the larger sized tile.

In section 5, we measure the efficiency of the algorithm using grind time (the time to update one cell one timestep). The effective grind time is computed as the total CPU time divided by the number of interior zones. Strictly speaking, almost all the work for updating a cell must also be done at the first ghost cell, so this measure overestimates the grind time. This is due to the fact that for n cells, there are $n + 1$ fluxes, which use information from both cells adjacent to it. The other ghost cells however are only used to calculate slope information and to add a little extra artificial viscosity for problems with very strong shocks. We investigated several approaches to eliminating the permanent storage of these ghost cells, or alternatively, using slightly larger grids so that the interior grid mapped to the fixed size tile, with borders that wrapped around the processor array. However, given the restricted layout and alignment directives, these approaches were much slower than the 30 to 40% penalty paid by not using the ghost cell processors for the bulk of the computation.

This remains a problem with an unsatisfactory solution.

There are two ways we take advantage of the additional communication on the CM-2 beyond treating it as a mesh machine: we use Fastgraph and the power of 2 NEWS wires. Although the calculation is adaptive, with most of the communication patterns changing dynamically through the course of the computation, there is a pattern that does not depend on grid placement and can be precomputed. The coarse grid is stored in memory using the permuted, or shuffled memory layout described in section 3. However, every now and then the coarse grid itself is integrated. For this step the solution is unshuffled, and then reshuffled at the end of the integration step. Regardless of how many levels there are, or where the grids are located, the permutation is identical (modulo the periodic offset). We use the communication compiler Fastgraph [12] to precompute an optimized routing for the shuffling operation and its inverse. Fastgraph itself is expensive to use, but it is only called once, it depends only on the refinement ratio, and the routing may be saved between runs, so we do not include Fastgraph in our run time results. Fastgraph saves a factor of 2 in the shuffling time over using the router; this is approximately 5% of the total run time on a typical run. Other users have reported much larger time savings when using the communication compiler; we surmise that because the permutation itself is so simple, the router does a good job of it to begin with. Nevertheless, for other classes of architectures besides a CM-2, and other applications besides AMR, an optimized preconfigurable routing would be extremely useful.

The second way we use the CM-2 as more than a mesh machine is with shifts (CSHIFT, EOSHIFT) of more than 1 location at a time. This is useful in the updating step of the AMR algorithm. For example, to compute the average of a 4 by 4 sub-block of fine grid cells so that the result may be injected onto the coarse grid, we sum by using a shift of one and then two in each direction. In the reverse procedure, the sum is logarithmically spread back to all 16 fine cells, so that the one lying on top of the coarse cell can do the actual injection. This reduces the number of shifts from 3 to 2 in each dimension for the sum and the spread, with the run time for this routine reduced by 1/3 as well. Even the NEWS network communication is much more expensive than floating point operations, and the updating routines involve very little computation. We also experimented with segmented scans (adds and copies), but found our implementation faster for such small segment lengths as 2 or 4.

5 Numerical Results

We present two types of measurement to indicate the performance of the AMR algorithm on the CM-2. First, we demonstrate the performance of the integrator as a function of grid size, without any adaptivity. The performance of the overall algorithm can be no better than this, since we count all other CPU time as overhead in the algorithm. Next we show

PatchSize	1 by 1		2 by 2		4 by 4	
	Integ. time	BC time	Integ. time	BC time	Integ. time	BC time
32	1.93	.96	7.76	2.27	30.69	3.07
64	5.40	3.25	21.55	7.56	86.45	19.34
128	18.60	7.34	74.55	29.05	298.82	73.41
256	70.63	20.87	281.56	73.84	1125.93	289.27

Table 3: CPU times with no refinement as a function of grid size and number of grids

numerical results and timings for experiments with 3 levels of grids. All the timings use the CM timer with version 1.1 of the slicewise compiler with optimization, on one sequencer attached to 8K processors of the CM-2 at NASA Ames Research Center. Our results show only the CM CPU time; the elapsed time (the sum of CPU and idle time) varies greatly, depending on how heavily loaded the CM/front end is. Our best idle times are typically between 2 and 5% of CPU time, depending slightly on the grid size.

Table 3 shows the CPU time for the integrator and the periodic boundary condition routine as a function of tile size. For this experiment we do not use any refinement (i.e. only coarse grids at the base level), but we do use several grids at that level. Notice that for patch sizes of 64 or more, doubling the number of points in each dimension gives integration times that increase by a factor of $18.60/5.40 = 3.4$, and $70.63/18.60 = 3.7$, unfortunately less than, though close to 4. When we go from a patch size of 32 to 64, this is far from the case ($5.40/1.93 = 2.80$). Although we would prefer to use the smaller patch size for greater efficiency in resolving localized flow features in the solution, the extra cost of integration on such a small patch makes this choice infeasible, at least for the current compilers. For 3 dimensional calculations, we expect this to change.

The time for the periodic boundary conditions is sublinear. With one coarse grid, periodic boundary conditions must be applied at all four sides of the grid. This type of boundary condition is the most expensive since data must be shifted approximately halfway across the computational domain due to the ghost cells. (Reflecting wall boundary conditions are less expensive, but since they must be computed inside the integrator , the interpretation of the integration run-time as a function of tile size becomes more complicated.) When there are several coarse grids in the computational domain, each grid also gets (interior) boundary conditions from the adjacent grids, which is an efficient operation.

The grind times for this problem (microseconds per cell per update) are summarized in table 4. For a given tile size, the grind times decrease due to the decreasing cost of the periodic boundary conditions. For a fixed number of base grids, (looking down a column), the grind times decrease due to the better performance of the CM-2 on larger blocks of data. We feel that reasonable performance is possible with 64 sized patches, and use that

PatchSize	1 by 1	2 by 2	4 by 4
32	52	45	38
64	28	24	22
128	19	19	17
256	15	15	15

Table 4: μ secs to update one zone for one timestep (includes all overhead) as a function of tile size

size in our the two dimensional calculations with adaptive refinement. Although the grind time for a 64 sized patch is 50% more than the grind time for a 256 sized patch, the latter has 16 times the number of points. The total cost of the computation will be cheaper using smaller sized patches as long as there are fewer than 8 of them. (We hope to improve this break even point with future release of the compiler).

We wrap up this section by describing a calculation modeling the flow of a hot dense gas leaving a square trench into a low density and temperature medium. This calculation is a prototype for modeling laser deposition of energy into an integrated circuit substrate. Ultimately lasers will be used to dig micron scale trenches in integrated circuits. Before this can be done, it is important to understand the dynamics of the laser induced flow so that debris patterns can be categorized or even predicted as a function of energy deposition.

The setup of this problem is straightforward. The entire computational region is embedded in a unit square. Within this region, a box of size 1/8 wide by 7/8 height is cut out of the computational domain from the upper left corner. This rectangle is a void region, meaning it is not part of the computational region, but is part of the solid wall exterior boundary. All boundaries of the computational domain are set to be reflecting boundaries, so no fluid escapes from the computational region. The 1/8 by 1/8 region in the lower left has initial density of 1.0 and pressure 10.0. The 7/8 wide by 1 high region in the right part of the domain, called the ambient region, has initial density and pressure set to 0.1 and 1.0 respectively. The velocities everywhere in the domain are initially set to zero and an ideal equation of state with a γ of 1.4 is used to relate density, internal energy and pressure. An illustration of the problem setup is shown in figure 4.

The major feature of the flow at late time (illustrated in figure 5) is a large bow shock penetrating into the ambient fluid. Even at this early time, the initial conditions look like a point source for the bow shock as the shock is already quite spherical. Behind the shock is the contact discontinuity delineating the boundary between the hot gas and the ambient gas. As the hot gas flows out into the ambient region, a shear layer forms causing the vortical rollup of the gas. The size of the rollup region will play an important part in the size of the debris regions around the trench.

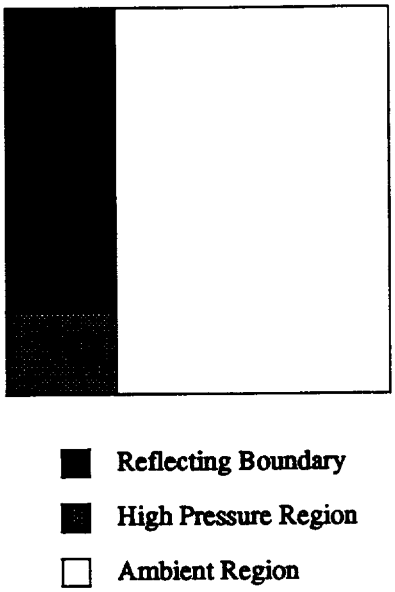
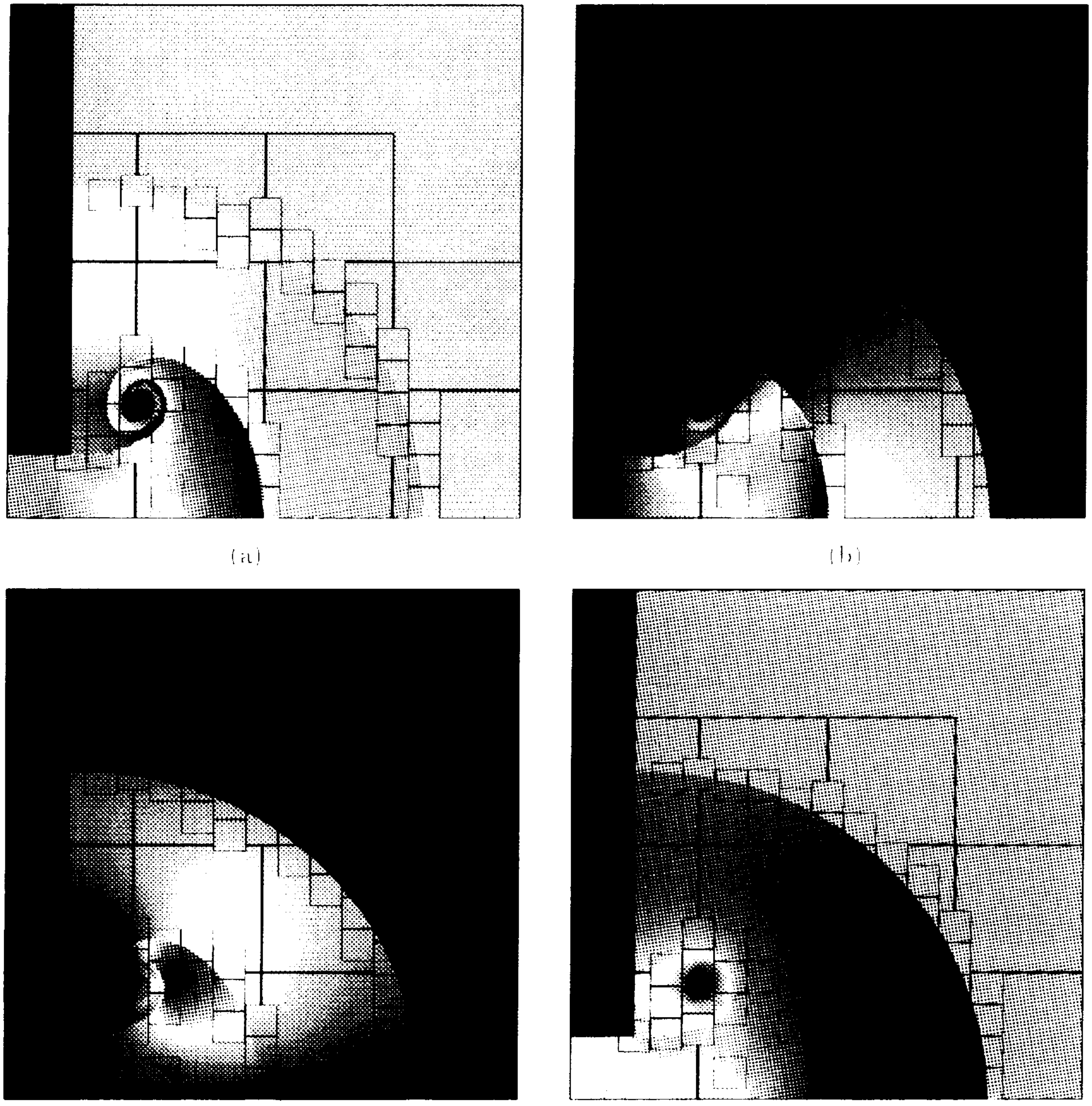


Figure 4: A schematic of the initial conditions for the prototype calculation modeling laser deposition of energy into an integrated circuit substrate.

Table 5 is a breakdown of CPU times for the AMR code when run for ten coarse grid time steps. The bulk of the time taken outside of the integration routine is taken by the boundary conditions routines. Within this category, 72 % of the time is taken interpolating from coarse grids to fine grids. The grind time for this calculation is 25.1 microseconds/cell which fits within the bounds shown in table 4. Although the grind time is a little under a factor of two for the best uniform case (256 x 256), the savings in computation costs is still great as only a fraction of the computation region is computed using a fine mesh. The number of cells advanced by the AMR algorithm is only 12% of the number of cells that would need to be advanced by the uniform case. Therefore a factor of four overall gain in efficiency is achieved.

6 Conclusions

Despite the disappointing performance of the CM-2 for small grid sizes, we feel that the two dimensional implementation of AMR is a useful tool. Our overall performance on an 8K CM-2 is roughly equivalent to a single head of the Y-MP. We hope in the future that richer array constructs, and layout and alignment directives will make possible further improvements in the efficiency. Although our current choice of tile size does not scale to use the full machine in an efficient manner, we believe that three dimensional calculations,



(a) (b) (c) (d)

Figure 5: Prototype integrated circuit problem results at time .124 and cycle 70. Figure (a) is a raster plot of the logarithm of density, (b) and (c) are plots of x and y momentum respectively, and (d) is a plot of the logarithm of total energy. Values range from the low color blue (cold) to the high color red (hot).

ORIGINAL PAGE
COLOR PHOTOGRAPH

Routine	CPU Time	Percent
Integration	587.79	76%
Boundary Cond.	107.35	14%
Cell Updates	10.92	1%
Error Est./Regrid	49.01	6%
Flux Correction	40.96	5%
Fastgraph	23.03	-

Table 5: Distribution of CPU usage by category within the AMR code for the integrated circuit calculation.

(or improved two dimensional calculations with tile size 32) would scale to use the full Connection Machine. Research along these directions is in progress.

Our greatest disappointment in working on the CM-2 came from its inability to exploit a coarser grained parallelism we found, on top of the fine grained data parallelism that is the basis of our approach. The source of this additional parallelism is the multiple grids at any given level. For example, instead of integrating one grid at a time, integrate all grids at a given level. Exactly the same operations are done to integrate any grid. We were particularly excited about the possibility of parallelizing the boundary condition routines. Ghost cells might possibly cause a load imbalance, but the periodic offsets in the layout of the grids would mitigate this by distributing the locations of the ghost cells in memory. When grids are shifted, the ghost cells often end up interior to the grid, rather than around the perimeter of the grid. Figure 6 illustrates this schematically.

Unfortunately, version 1.1 of the compiler does not parallelize (vectorize) across serial dimensions, and all our attempts to restructure the do loops to force it to do so failed. By integrating (interpolating, updating, etc.) many patches at a single time, we feel the 32 sized tile would be practical, allowing greater overall efficiency in the AMR algorithm. In fact, when a fully operational version of the CM-5 becomes available this coarser grained parallelism would make an interesting case study in a hybrid SIMD/MIMD model of computation.

Acknowledgements. It is a pleasure to acknowledge discussions with Scott Baden, Yanmin Sun, and Steve Hammond during the course of this work. M. Berger was supported in part by DOE grant DE-FG02-88ER25053, by AFOSR grant 91-0063, and by a PYI, NSF ASC-8858101. Additional support was provided in part by Cooperative Agreement NCC2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center. J. Saltzman was

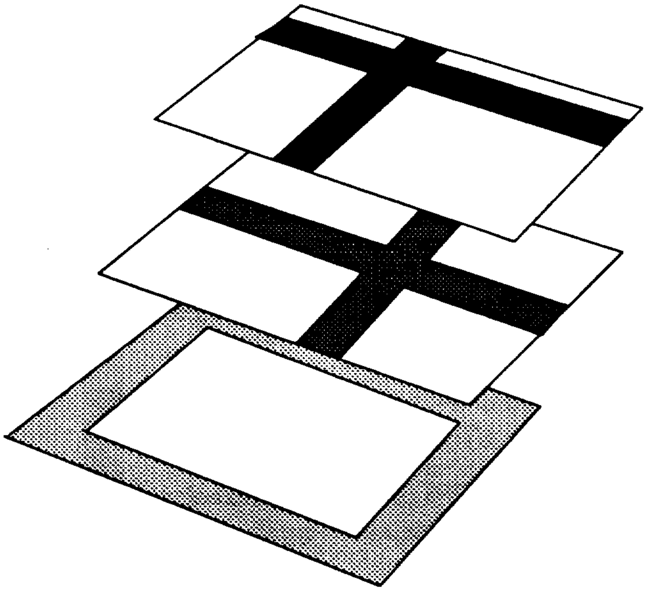


Figure 6: The boundary ghost cells are shaded on each grid patch. They are distributed to a different memory location in each patch.

supported by the U.S. Department of Energy through Los Alamos National Laboratory under contract No. W-7405-ENG-36 with additional support from the Center for Research in Parallel Computation and DARPA/NSF grant DMS-8919074 UC-Berkeley. We would like to thank the NAS systems division at NASA Ames for access to their Connection Machine CM-2.

References

[1] J. Bell, M. J. Berger, J. Saltzman, and M. Welcome. Three dimensional adaptive mesh refinement for hyperbolic conservation laws. Preprint UCRL-JC-108794, Lawrence Livermore National Laboratory, Livermore, CA, December 1991.

[2] M. J. Berger. Adaptive mesh refinement for parallel processors. In *Proceedings of 1987 SIAM Conference on Parallel Processing*, Los Angeles, CA, December 1987.

[3] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[4] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:482–512, 1984.

[5] R. Biswas. *Parallel and Adaptive Methods for Hyperbolic PDES.* PhD thesis, RPI, 1991.

[6] R. Biswas and J. Flaherty. Parallel and adaptive methods for hyperbolic partial differential equations. In *Proceedings of the 7th IMACS International Conference on Computer Methods for Partial Differential Equations*, New Brunswick, NJ, June 1992.

[7] J. Cerutti and H. Trease. The free-Lagrange method on the Connection Machine. In W. C. H. Trease, J. Fritts, editor, *Proceedings of the Next Free-Lagrange Conference*, Copper Mountain, Colorado, 1991.

[8] P. Colella. Multidimensional upwind methods for hyperbolic conservation laws. *Journal of Computational Physics*, 87:171–200, 1990.

[9] P. Colella and P. Woodward. The piecewise parabolic method (ppm) for gas-dynamical simulations. *Journal of Computational Physics*, 54(1):174–201, April 1984.

[10] W. Crutchfield. Load balancing irregular algorithms. Preprint UCRL-JC-107679, Lawrence Livermore National Laboratory, Livermore, CA, July 1991.

[11] W. Crutchfield. Parallel adaptive mesh refinement: An example of parallel data encapsulation. Preprint UCRL-JC-107680, Lawrence Livermore National Laboratory, Livermore, CA, July 1991. Submitted to Concurrency, Practice and Experience.

[12] E. D. Dahl. Mapping and compiled communication on the Connection Machine system. In *Proceedings of The Fifth Distributed Memory Computering Conference*, April 1990.

[13] W. Gropp and D. Keyes. Semi-structured refinement and parallel domain decomposition methods. In J. S. P. Mehrotra and R. Voigt, editors, *Unstructured Scientific Computation on Scalable Multiprocessors.* MIT Press, 1992.

[14] S. W. Hammond and T. J. Barth. An efficient massively parallel Euler solver for unstructured grids. *AIAA Journal*, 30(4):947–952, April 1992.

[15] L. F. Henderson, P. Colella, and E. G. Puckett. On the refraction of shock waves at a slow-fast gas interface. *Journal of Fluid Mechanics*, 224:1–27, 1991.

[16] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[17] Y. Kallinderis and A. Vidwans. Parallel adaptive-grid navier-stokes algorithm development and implementation. Technical report, University of Texas at Austin, 1992. Preprint.

[18] O. Lubeck, M. Simmons, and H. Wasserman. The performance realities of massively parallel processors: A case study. Preprint LA-UR-92-1463, Los Alamos National Laboratory, Los Alamos, NM, 1992. Submitted to IEEE Supercomputing '92.

[19] S. McCormick and D. Quinlan. Dynamic grid refinement for partial differential equations on parallel computers. In *Proceedings of 7th International Conference on Finite Element Methods in Flow Problems.* University of Alabama, April 1989.

[20] J. Quirk. *An adaptive grid algorithm for computational shock hydrodynamics.* PhD thesis, College of Aeronautics, Cranfield Institute of Technology, 1991.

[21] M. Ruffert. Collisions between a white dwarf and a main-sequence star II. Simulations using multiple nested refined grids. Preprint MPA 657, Max Planck Institute for Astrophysics, Garching, March 1992. Submitted to Astronomy and Astrophysics.

[22] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, October 1991.